#### *Programación con sockets: respuestas*

4. TCPEchoClient4.c y TCPEchoServer4.c

* 1. ¿Se podría hacer que el servicio escuche en más de un puerto?
     1. Por la forma en que está programado no se puede hacer, ya que “accept” es bloqueante. Y en struct sockaddr\_in no es posible indicar más de un puerto
  2. Suponiendo que la red local es 192.168.1.0/24 ¿Se podría modificar para que el servidor sólo escuche pedidos de la red local? Si la respuesta es no, ¿podría sólo atender los pedidos provenientes de la red local?
     1. Se puede modificar para atender sólo los que tengan como destino una de las IPs asignadas al servidor, pero no se puede indicar a TCP que sólo acepte pedidos de conexión en base al origen. Para atender pedidos únicamente provenientes de la red local se debe hacer el filtro a nivel de la aplicación: una vez aceptada una conexión analizar el contenido de la variable clnAddr, de la cual podemos obtener la dirección IP de origen. O recurrir a servicios externos al servidor ECHO, más precisamente un firewall.
  3. ¿Qué sucede si un cliente no cierra la conexión?
     1. Como se mencionó en la clase, el servidor se queda esperando el cierre, lo cual provoca un DOS (*Denial of service*), ya que nadie más se podrá conectar.
  4. ¿Qué sucede si el servidor en la respuesta envía octetos de menos?
     1. El cliente se queda esperándolos y nunca cierra la conexión
  5. ¿Qué sucede si el servidor en la respuesta envía octetos de más?
     1. El cliente los ignora y cierra la conexión
  6. Modificar el protocolo (y por lo tanto la implementación del cliente y servidor) para que el servidor, antes de leer lo que el cliente envía, le envíe un mensaje de bienvenida. Una vez que el cliente leyó el mensaje de bienvenida envía su mensaje y espera por la respuesta.
     1. El servidor inmediatamente después del accept debe escribir en el socket el mensaje de bienvenida. El cliente una vez establecida la conexión primero lee el mensaje (debe saber dónde termina o su longitud) y luego envía.
     2. ¿Puede el servidor decidir en forma arbitraria el mensaje o debe haber un acuerdo previo sobre las características del mismo?
        1. De alguna forma el cliente tiene que saber hasta dónde leer, por lo que el mensaje debe tener una longitud fija o estar marcado el final, por ejemplo que el mensaje de bienvenida sea una línea de texto terminada en \r\n.
  7. Modificar TCPEchoClient4.c para que use IP y puerto específico de origen (usar bind()). Ver qué sucede si elige una IP inválida o un puerto en uso.
     1. Probarlo
  8. ¿Qué sucedería si un servidor TCP en vez de usar un socket pasivo para atender conexiones y un socket activo para usar esa nueva conexión, pudiera usar un único socket? Esto es, que pudiera crear un *socket server*, llamar a bind() y listen(), esperar por una comunicación, enviar y recibir a través de ese socket y cuando termine cierre y repita el proceso?
     1. Con el esquema de servidor iterativo planteado no habría ninguna diferencia. Pero sería un esquema que no permitiría tener más de una conexión simultánea

6. UDPEchoClient.c y UDPEchoServer.c

* 1. ¿Qué cambios habría que hacer si quisiéramos aceptar mensajes arbitrariamente largos? Asumir que no habrá pérdida y los datagramas se reciben en forma ordenada.
     1. En el servidor no habría que hacer ningún cambio, ya que en vez de estar haciendo ECHO por un mensaje completo estaría haciendo ECHO por “partes” del mensaje total. En el cliente sí habría que hacer cambios: tendría que enviar y recibir tantos datagramas como sea necesario. Asumiendo que no hay pérdida y se reciben en orden ese sería el único cambio, si asumimos pérdida y/o llegada en distinto orden sería mucho más complejo (recordar cómo brinda confiabilidad TCP)
  2. ¿Sería correcto usar este cliente en producción? Pista: qué sucede si el mensaje enviado o la respuesta se pierde.
     1. No, ya que el cliente se queda esperando una respuesta que tal vez no llegue nunca. Debería establecer un timeout y decidir luego de ese timeout qué hacer: reenviar, darse por vencido, etc.
  3. ¿Cómo se podría usar estos códigos para determinar empíricamente el tamaño máximo de un datagrama? Determinar si el máximo es el mismo para IPv4 e IPv6.
     1. Se debe armar un mensaje con un tamaño inicial, si UDP lo acepta, incrementar el tamaño y volver a intentar. SI el sendTo falla (retorna -1 y errno es **EMSGSIZE**), decrementar el tamaño hasta que no falle.
  4. ¿Qué sucede si el servidor envía menos bytes de los recibidos? ¿Y si envía más?
     1. En este caso el cliente verifica que la longitud del mensaje sea igual a la longitud enviada y en caso contrario informa del error
  5. Pensar qué cambios habría que hacer si se quieren enviar y recibir mensajes arbitrariamente largos y que el protocolo sea confiable.
     1. Como habría que enviar varios datagramas, se debe implementar a nivel de aplicación los mecanismos que implementa TCP para brindar confiabilidad, por ejemplo
        1. Recibir un ACK por cada datagrama enviado. Para lo cual cada datagrama además de los datos debe tener una identificación.
        2. Almacenar en algún buffer cada datagrama enviado hasta que se reciba el ACK
        3. Por cada datagrama enviado establecer un timeout, y en caso de no recibir el ACK volver a enviarlo (con el mismo ID para que destino verifique si llega duplicado)
        4. Almacenar los datagramas que lleguen en desorden, a la espera de los anteriores, para mostrar la información en forma ordenada. Se podría usar otro campo para numerar los datagramas o que el ID tenga un orden
        5. Contemplar que un datagrama puede ser recibido dos o más veces (por ejemplo porque se perdió el ACK del mismo)

7. Más allá de las críticas que se pueda hacer al protocolo (y a las aplicaciones sobre horóscopos en general), lo que se pide es críticas a la implementación. El error más grave cometido es considerar una secuencia “ideal” entre el servidor y el cliente en la cual el primer datagrama recibido corresponde a un pedido de horóscopo y el siguiente datagrama corresponde a un ACK del mismo cliente, como si estuviéramos usando TCP u otro protocolo de transporte orientado a conexión. Así como los datagramas se pierden, también puede ser que cuando esperamos por un ACK en realidad sea el pedido de otro cliente, y que al esperar un signo en realidad se reciba un ACK. En caso de insistir en implementar este protocolo y hacerlo con UDP lo correcto sería hacer una lectura de un datagrama, si corresponde a un signo, responderle y guardar en un “conjunto de pendientes” el IP, si el datagrama corresponde a un ACK, buscarlo en el “conjunto de pendientes” y darle el tratamiento adecuado